**Amendments to the Specification:**

Replace paragraphs [0002], [0005]–[0007], [0009], [0014]–[0019], [0021]–[0023], [0025], [0027], and [0029]–[0034] in their entireties with the paragraphs shown below.

**[0002]** In a graphics processing system, objects to be displayed are generally represented by a collection of polygons. Polygons are generally chosen due to the existence of efficient algorithms for the rendering [[of]] thereof. However, frequently the object that is approximated by polygons is really a curved shape. Most methods for describing these surfaces fall under the classification of "higher order ~~surfaces~~ surfaces" (HOS).

**[0005]** As primitive may be deeper within the rendered scene, such as having a smaller depth value, the ratio of pixels per primitive may be reduced. During the rendering of the pixels, the depth offset may adversely ~~effect~~ affect the computation of tessellation factors. Using an adaptive tessellation technique, various adaptive tessellation factors may be computed using a software algorithm. Although, the technique includes processing time limitations and further includes space limitations as there must be physical space within a graphics processing system for a processor, such as a general purpose processor, executing operating instructions for calculating the adaptive tessellation factors.

**[0006]** Therefore, during the processing of tessellation factors, this technique may be inefficient as ~~requiring~~ it requires the offloading of vertex information, the computation of the adaptive tessellation factors and ~~the~~ loading these tessellation factors back into the processing pipeline.

**[0007]** Another approach utilized for generating tessellation factors is an independent hardware configuration, which interfaces with the graphics rendering pipeline. This hardware approach provides a separate processor in communication with the graphics rendering pipeline, ~~wherein this approach~~ and thus not only requires valuable ~~real estate~~ real estate for the extra

hardware, but ~~includes~~ also requires further processing time for the transmission of primitive indices to the hardware device and the transmission of vertex tessellated data back to the processing pipeline.

**[0009]** As such, there exists a need for generating tessellation factors independent of the primitive types, wherein the tessellation factor generation efficiently uses existing available processing resources and without adversely ~~effecting~~ affecting system processing speeds.

**[0014]** FIG. 4 illustrates a flow chart of the steps of a method for dual pass adaptive tessellation in accordance with one embodiment of the present invention; and

**[0015]** FIG. 5 illustrates a flow chart of the steps of another method for dual pass adaptive tessellation in accordance with another embodiment of the present ~~invention; and~~ invention.

**[0016]** ~~FIG 1~~ FIG. 1 illustrates an apparatus 100 for dual pass adaptive tessellation. The apparatus 100 includes a memory interface 102, a global register bus 104, and a vertex grouper tessellator 106 having a direct memory access engine 108 dispose therein. The apparatus 100 further includes a shader processing unit 110, a plurality of vertex shader input staging registers 112 coupled to a plurality of vertex shaders 114 and a memory 116 coupled to the shaders 114 across a bus 118. The shader processing unit 110 may be a sequencer or a control flow processor. Moreover, each of the shaders 114 includes a math processing unit, which may be any suitable processing unit capable of executing mathematical operations.

**[0017]** In one embodiment, the vertex grouper tessellator 106 is pass-through enabled for a first pass. During the first pass, an index list 120 is provided from the memory interface 102 to the direct memory access engine 108. Furthermore, vertex grouper tessellator state data, direct memory access request data and a draw initiator 122 are received from the global register bus 104 by the vertex grouper tessellator 106. The indices of primitives are sent through the vertex

grouper tessellator 106 to be loaded into registers of shader pipes, such as registers 112 and shaders 114. The vertex grouper tessellator 106 also sends an auto-index value for each primitive to one of the shader registers 112. Most specifically, the vertex grouper tessellator 106 provides primitive information 124 to the shader sequencer 110 which thereupon distributes partitioned information 126 to each of the vertex shader input state registers 112.

**[0018]** The vertex grouper tessellator 106, during a first pass, also generates a per-process vector 130, [[a]] per primitive data 132 and [[a]] per packet data 134. During the first pass, no pixels are generated, but the signals 130-134 may still have to pass synchronization signals for vertex and state deallocation.

**[0019]** Referring back to the shader pipeline, the vertex shaders 114 receive vertex data arrays 134 retrieved from the memory 116. The shader processing unit 110 fetches corner vertices of high order surface ~~primitive~~ primitives and computes tessellation factors based on corner vertices of edges of primitives. In another embodiment, during the first pass, the vertex shaders 114 could also generate control points for high order surfaces. The tessellator 106 is disabled during this pass.

**[0021]** During the second pass, more than one ~~indices~~ index can be passed to the vertex grouper tessellator 106 along with primitive types such as tri-list, quad-list, line-list, rect-patch, line-patch, or tri-patch. The vertex grouper tessellator ~~106, generates~~ 106 generates the parametric ~~coordinance~~ coordinates (u, v) for tensor products surfaces or ~~bary-centric~~ barycentric coordinates (u, v, w) for triangular surfaces using tessellation factors computed during the first pass. These coordinates, along with original indices and/or an auto-index value, are passed to the shader register 112 for evaluating ~~tessellating~~ tessellated vertices based on user evaluation

shaders. The tessellation engine 100 also generates sub-primitive information for clipper and primitive assemblers for paths 134, 132, 130.

**[0022]** ~~Fig. 2~~ FIG. 2 illustrates the apparatus 100 through a second pass, with the vertex grouper tessellator 106 being enabled. The memory interface 102 receives a plurality of tessellation factors 150 received from the memory 116 of ~~Fig. 1~~ FIG. 1. Using the direct memory access engine 108 within the vertex grouper tessellator 106, the tessellation factors are retrieved by a direct memory access request 152, wherein the tessellation ~~factor~~ factors are provided for the vertex grouper tessellator 106 operating ~~with~~ in an adaptive mode. In this second pass, the tessellation factors 150 are accessed as indices and an auto index is generated, wherein the auto index 124 is loaded to the registers 112 of the shaders 114. A primitive mode indicator set to a tessellation mode such that the indices that go to the tessellation engine are interpreted as tessellation factors. The vertex group tessellator 106 generates ~~bary-centric~~ barycentric or tensor coordinates which are loaded to the shader registers 112 along with the auto-index value. The shaders 114 compute the tessellated vertices by fetching control points from locations specified by the auto-index value and using ~~bary-centric~~ barycentric or tensor product coordinates.

**[0023]** In one embodiment, a primitive data array 154 is retrieved from memory such that the vertex shaders 114 may be utilized for the generation of newly computed sub-primitive information 156.

**[0025]** Therefore, by using a dual-pass system for adaptive tessellation, the vertex grouper tessellator 106 is disabled in a pass-through mode for generating tessellation factors using the shaders 114 during pass one and the tessellator part of vertex grouper tessellator 106 is enabled in a tessellator mode for the second pass such that the tessellation factors themselves are used to

generate tessellation outputs such as the per process vector output, the per primitive output and the per packet output, in conjunction with the ~~sub primitive~~ sub-primitive data 156. As such, the present invention utilizes a single tessellation engine 100 and reuses the same engine with different primitive types to generate the needed output. Therefore, the present invention not only reduces the amount of real estate needed for hardware components, the engine 100 improves processing efficiency by allowing for the generation of tessellation output within the pixel/vertex processing pipeline.

**[0027]** In one embodiment, the tessellation engine 100 is a fixed-function block. This engine 100 understands a finite set of modes, each of which is defined for a specific purpose. The below table shows all of the modes supported by the present invention. As recognized by one having ordinary skill in the art, the below table indicates various modes for exemplary purposes only and illustrates one embodiment of the present invention. As recognized by one having ordinary skill in the art, any other suitable implementation in accordance with the information listed below for adaptive tessellation may also be suitably implemented [[and]] within the scope of the present invention.

**[0029]** FIG. 3 illustrates an exemplary embodiment of a data field 200 which may be stored in the memory 116 of FIG. 1. The data 200 includes a tessellation factor field 202 storing three tessellation factors 204 for a triangle primitive type and an original indices field 206 storing original indices 208. The data field 200 further includes a base address 210 for reference of the tessellation factor therefrom. The tessellation ~~factor~~ factors 202 and the original indices 206 may have a data size relative to the corresponding number of vertex shaders 114 of FIG. 2 such that the associated tessellation information may be generated therefrom. It is recognized by one

having ordinary skill in the art and any other suitable memory data structure may be implemented and that the data structure 200 of FIG. 3 is for illustration purposes only.

**[0030]** FIG. 4 illustrates the steps of the method for dual pass adaptive tessellation. The method begins, step 220, by receiving vertex information and an index list, ~~one~~ wherein the index list is received from a memory device in step 222. With reference to FIG. 1, index list 120 is retrieved from a memory interface across a direct memory access engine 108. The next step, step 224, consists of generating primitive indices from the vertex information and an auto-index value for each of the primitive indices. Also from the first pass, the next step is generating a plurality of shader sequence ~~outputs, 226~~ outputs 226. Illustrated with reference to FIG. 1, the plurality of shader sequence outputs 126 are provided to [[in]] the plurality of vertex shader input staging registers 112, step 228. Therefore, within the first pass, a plurality of tessellation factors are generated in response to shader sequence output, step 230.

**[0031]** The vertex shader ~~sequence~~ output 136, in one embodiment, is stored in a memory ~~location~~ such that ~~they~~ it may be further received as a plurality of indices, step 232, during a second pass. As illustrated in FIG. 2, the tessellation factors 150 are provided to the memory interface 102 such that they may be retrieved from the direct memory access engine 108. As such, one embodiment of the method is complete, step 234.

**[0032]** FIG. 5 illustrates another embodiment of a method for dual pass adaptive tessellation. The method ~~begin~~ begins, 250, by performing the steps during a first pass of the steps 222 through ~~steps~~ 230 of FIG. 4, step 252. As discussed above, the steps include receiving primitive information and an index list, and an auto-index value for each of the primitive indices, generating a plurality of shader sequence ~~output~~ outputs, providing the shader sequence ~~output~~ outputs to a plurality of vertex shader input staging registers and generating a plurality of

tessellation factors in response to the shader sequence outputs. The next step, step 254, is writing the plurality of tessellation factors to a memory device. As discussed above with regards to FIG. 1, the memory 116 may be any suitable memory device capable of storing and allowing for the retrieval of the tessellation factors therefrom. Steps 252 and 254, in the present embodiment, occurred during a first pass of the tessellation engine 100 of FIGS. 1 and 2.

**[0033]** During a second pass, the first step, 256, is receiving the tessellation factors from the memory device. The next step, step ~~258 is~~ 258, is generating an auto-index value for each of the plurality of indices. The next step, step 260, is generating a plurality of ~~bary-centric~~ barycentric coordinates based on the tessellation factors, or in another embodiment, tensor product coordinates may be computed based on primitive type. As discussed above, the coordinates may be (u, v) or (u, v, w) based on the higher order surface.

**[0034]** Still within the second pass, the next step, step ~~262 is~~ 262, is computing a plurality of tessellated vertices by fetching a control point specified by the auto-index value for each of the plurality and indices. In one embodiment, the control point specified by the auto-index value for each of the plurality of indices may be disposed within the memory 116 illustrated in FIG. 1 and provided to the vertex shaders 114. As such, output signals 130, 132, and 134 generated by the vertex grouper tessellator 106 of FIG. 1 and ~~sub-primitive~~ sub-primitive data further generated by the vertex shaders 114 of FIG. 2.